Express.js 中文文档

wizardforcel

Published
with GitBook



目錄

Express.js 中文文档

来源：Express.js 中文文档

新手指南

开始

在确认已经安装了node之后(下载), 在你的机器上创建一个目录，让我们来开始你的第一个应用程序吧

```
$ mkdir hello-world
```

在这个目录中你首先得定义一下你的应用程序“包”文件，它和其它的node程序包是一样的。 你得在这个目录中创建一个package.json文件，在里面express作为一个依赖。 你也可以使用 `npm info express version` 来获取express最新的版本号， 最好使用最新的版本号而不是下面的3.x，这样新出的功能就不会让你感觉到奇怪了。

```
{
  "name": "hello-world",
  "description": "hello world test app",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "express": "3.x"
  }
}
```

现在package.json文件已经准备好了，使用 `npm(1)` 安装依赖， 这里的依赖仅仅是Express。

```
$ npm install
```

当npm完成后，Express 3.x 和它的依赖就安装到你的 ./node_modules 目录里了。你可以通过 `npm ls` 来确认一下，它会把Express 和它的依赖展示成下面的树状结构。

```
$ npm ls
hello-world@0.0.1 /private/tmp
└─┬ express@3.0.0beta7
  ├── commander@0.6.1
  ├─┬ connect@2.3.9
  │ ├── bytes@0.1.0
  │ ├── cookie@0.0.4
  │ ├── crc@0.2.0
  │ ├── formidable@1.0.11
  │ └── qs@0.4.2
  ├── cookie@0.0.3
  ├── debug@0.7.0
  ├── fresh@0.1.0
  ├── methods@0.0.1
  ├── mkdirp@0.3.3
  ├── range-parser@0.0.4
  ├─┬ response-send@0.0.1
  │ └── crc@0.2.0
  └─┬ send@0.0.3
    └── mime@1.2.6
```

现在我们来写真正的代码了！创建一个名为app.js 或者 server.js的文件，叫什么看你个人喜好了。 载入express 然后使用代码 `express()` 创建一个新的应用程序:

```
var express = require('express');
var app = express();
```

在这个应用程序实例里，你可以通过 `app.VERB()` 定义路由，下面的例子是"GET /"返回 "Hello World" 字符串。 `req` 和 `res` 对象是和node原生提供给你的一致的，你也可以执行 `res.pipe()` , `req.on('data', callback)` 等任何事情在没有Express的情况下可以做的事情。

Express 给这些对象加了一个封装好的方法，比如 `res.send()` , 它会帮你设置Content-Length:

```
app.get('/hello.txt', function(req, res){
  res.send('Hello World');
});
```

现在我们通过执行 `app.listen()` 来绑定并监听连接。 它接受的参数和nodenet.Server#listen()的方法一致:

```
var server = app.listen(3000, function() {
    console.log('Listening on port %d', server.address().port);
});
```

使用express(1) 来生成一个应用程序

Express团队维护了一个可以快速生成项目模板的可执行文件，这里命名为 `express(1)` . 如果你使用npm全局安装的express-generator, 在你的机器任何位置它都是可用的:

```
$ npm install -g express-generator
```

这个工具提供了一个非常简单的生成一个程序骨架的功能，但是它也有局限，比如它只支持很少的几个模板引擎。 而事实上Express几乎支持所有的为node所建的模板引擎。 使用 `--help` 查看一下帮助:

```
Usage: express [options]

Options:

  -h, --help          输出帮助信息
  -V, --version       输出版本号
  -e, --ejs           添加 ejs 模板引擎支持 (默认为jade)
  -H, --hogan         添加 hogan.js模板引擎支持
  -c, --css <engine>  样式 <引擎> 支持 (less|stylus) (默认为css)
  -f, --force         强制在非空目录执行</engine>
```

如果你想生成一个支持Jade, Stylus的应用程序，只需要简单的执行下面的命令：

```
$ express --sessions --css stylus --ejs myapp

create : myapp
create : myapp/package.json
create : myapp/app.js
create : myapp/public
create : myapp/public/javascripts
create : myapp/public/images
create : myapp/public/stylesheets
create : myapp/public/stylesheets/style.styl
create : myapp/routes
create : myapp/routes/index.js
create : myapp/views
create : myapp/views/index.jade
create : myapp/views/layout.jade

install dependencies:
  $ cd myapp && npm install

run the app:
  $ DEBUG=myapp node app
```

和其它node程序一样，你必须安装依赖：

```
$ cd myapp
$ npm install
```

然后让我们运行它吧！

```
$ node app
```

这些就是一个简单的应用程序创建和运行的所有步骤。 记住Express没有限定任何的目录结构，这只是一个方便你工作的基本结构。 如果你想得到更多怎么组织目录结构选择，可以查看github上的示例。

错误处理

错误处理的中间件和普通的中间件定义是一样的， 只是它必须有4个形参，这是它的形式： `(err, req, res, next)`：

```
app.use(function(err, req, res, next){
  console.error(err.stack);
  res.send(500, 'Something broke!');
});
```

一般来说非强制性的错误处理一般被定义在最后，下面的代码展示的就是放在别的 `app.use()` 之后：

```
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
app.use(function(err, req, res, next){
  // logic
});
```

在这些中间件里的响应是可以任意定义的。只要你喜欢，你可以返回任意的内容，譬如HTML页面, 一个简单的消息，或者一个JSON字符串。

对于一些组织或者更高层次的框架，你可能会像定义普通的中间件一样定义一些错误处理的中间件。 假设你想定义一个中间件区别对待通过XHR和其它请求的错误处理，你可以这么做：

```
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
app.use(logErrors);
app.use(clientErrorHandler);
app.use(errorHandler);
```

通常 `logErrors` 用来纪录诸如stderr, loggly, 或者类似服务的错误信息：

```
function logErrors(err, req, res, next) {
  console.error(err.stack);
  next(err);
}
```

`clientErrorHandler` 定义如下，注意错误非常明确的向后传递了。

```
function clientErrorHandler(err, req, res, next) {
  if (req.xhr) {
    res.send(500, { error: 'Something blew up!' });
  } else {
    next(err);
  }
}
```

下面的 `errorHandler` "捕获所有" 的异常， 定义为:

```
function errorHandler(err, req, res, next) {
  res.status(500);
  res.render('error', { error: err });
}
```

在线用户计数

这一小节我们讲解一个小而全的应用程序，它通过Redis记录在线用户数。 首先你需要创建一个package.json 文件，包含两个依赖, 一个是redis 客户端， 另一个是Express。 另外需要确认你安装了redis, 可以能过执行 `$ redis-server` 来确认：

```
{
  "name": "app",
  "version": "0.0.1",
  "dependencies": {
    "express": "3.x",
    "redis": "*"
  }
}
```

接下来你需要你创建一个应用程序，和一个redis连接：

```
var express = require('express');
var redis = require('redis');
var db = redis.createClient();
var app = express();
```

接下来是纪录用户在线的中间件。 这里我们使用sorted sets, 它的一个好处是我们可以查询最近N毫秒内在线的用户。 我们通过传入一个时间戳来当作成员的"score"。 注意我们使用 User-Agent 作为一个标识用户的id。

```
app.use(function(req, res, next){
  var ua = req.headers['user-agent'];
  db.zadd('online', Date.now(), ua, next);
});
```

下一个中间件是通过**zrevrangebyscore**来查询上一分钟在线用户。 我们将能得到从当前时间算起在60,000毫秒内活跃的用户。

```
app.use(function(req, res, next){
  var min = 60 * 1000;
  var ago = Date.now() - min;
  db.zrevrangebyscore('online', '+inf', ago, function(err, users){
    if (err) return next(err);
    req.online = users;
    next();
  });
});
```

最后我们来使用它，绑定到一个端口！这些就是这个程序的一切了，在不同的浏览器里访问这个应用程序，你会看到计数的增长。

```
app.get('/', function(req, res){
  res.send(req.online.length + ' users online');
});

app.listen(3000);
```

给Express加一层代理

在Express的前端使用一个反向代理，比如 Varnish 或者 Nginx是非常常见的,它不需要额外的配置。 在通过 `app.enable('trust proxy')` 激活了"trust proxy" 设置后， Express 就会知道它在一个代理的后面， `X-Forwarded-*` 必须被信任， 通常情况下这些头是很容易被伪装的。

使用了这个设置后会有一些很棒的小变化。 首先由代理设置的 `X-Forwarded-Proto` 会告诉程序它是https 还是http 。 这个值会影响 req.protocol.

第二个变化是 req.ip 和 req.ips 的值会被 `X-Forwarded-For` 列表里的地址取代。

调试 Express

Express使用 debug 模块 来输出信息。如果想看到这些信息，可以在运行你的程序时设置 `DEBUG` 环境变量为 `express:*` ,调试信息会输出在终端里 。

```
$ DEBUG=express:* node app.js
```

使用上面的方式运行 `hello world` 的例子，将会输出下面的内容

```
express:application booting in development mode +0ms
express:router defined get /hello.txt +0ms
express:router defined get /hello.txt +1ms
```

获取更多关于 `debug` 的信息，可以查看 debug 文档

问答

怎样定义模型?

Express 根本没有涉及到数据库，这个任务留给了第三方的node模块，有了第三方的模块基本上可以与任何数据库交互

怎样做用户认证?

这是另一个Express不会做的事情， 你可以使用任何你想用的认证方案，这里有一个简单的例子。

Express 支持哪个模板引擎?

任何遵守这样回调的 `(path, locals, callback)` . 为了统一模板引擎接口和缓存，推荐查看consolidate.js 寻找帮助. 有些没有列出来的模板引擎没准也支持Express.

我应该怎样组织我的程序结构?

事实上这个没有一个标准答案，这与你的程序规模和团队强烈相关。 为了尽可能的灵活，Express没有规定程序的结构

你可以把路由和其它的一些程序特定的逻辑代码以任意的目录结构任意数量的文件存放。 查看下面的例子找点灵感

- Route listings
- Route map
- Route bootstrapping
- MVC style controllers

已经存在的简化这些模式的第三方Express扩展:

- Resourceful routing
- Namespaced routing

我应该怎样从多个目录提供静态文件服务?

你可以会在你的程序中多次使用任意一个中间件。 使用下面的方式，当你请求"GET /javascripts/jquery.js" 时，会先检查 "./public/javascripts/jquery.js", 如果它不存在，随后的中间件会检查 "./files/javascripts/jquery.js".

```
app.use(express.static('public'));
app.use(express.static('files'));
```

怎样在提供静态文件服务的时候加一个前缀路径名?

Connect's的中间件绑定技术允许你指定一个路径名前缀, 一个常用的例子是你可以前缀一个根本不是请求路径中一部分的字符。 假设你要请求 "GET /files/javascripts/jquery.js", 你可以把中间件挂在 "/files", 暴露出 "/javascripts/jquery.js"作为 `req.url` 来让中间件为这个文件提供服务:

```
app.use('/public', express.static('public'));
```

怎么迁移 **Express 2.x** 应用程序?

Express 2x甚至能支持到node 1.0, 所以可能没有必要由于Express 3x的重构和API改变就迁移, 如果你对2x感觉良好, 那就停留在那个版本上。真的要迁移可以看这里 或者查看一下3.x的改动列表

怎么处理**404s**?

在Express里404s不被认为是出错的结果, 所以错误处理中间件不会捕获404s,这是因为一个404只是由于有一些额外的工作没有做, 换而言之, Express已经执行了所有的中间件 / 路由分发, 然而没有发现有返回。你所要做的仅仅是在代码底部加一个中间件去处理没有返回的情况, 并且手动返回一个404

```
app.use(function(req, res, next){
  res.send(404, 'Sorry cant find that!');
});
```

Express里怎样处理异常?

定义错误处理的中间件跟定义普通的中间件没有什么区别, 仅仅是参数必须定义为4个, 它们定义如下 `(err, req, res, next)` :

```
app.use(function(err, req, res, next){
  console.error(err.stack);
  res.send(500, 'Something broke!');
});
```

查看 错误处理, 获取更多信息

怎样输出纯**HTML**文件?

不要这么做！根本没有必要直接使用 `res.render()` 输出HTML文件, 如果你有一个特定的文件应该使用 `res.sendfile()` ,如果你要使用一个目录里大量的静态资源提供服务, 请使用 `express.static()` 中间件.

Express的代码库有多大？

Express 是一个非常小的框架，3.0.0正式发布版只有932行源码，Express强烈依赖的Connect只有267行源码，Connect可选的中间件和扩展总共1143行源码，并且只有到使用时才会加载

API 参考

express()

创建一个express应用程序

```
var express = require('express');
var app = express();

app.get('/', function(req, res){
  res.send('hello world');
});

app.listen(3000);
```

Application

app.set(name, value)

将设置项 `name` 的值设为 `value`

```
app.set('title', 'My Site');
app.get('title');
// => "My Site"
```

app.get(name)

获取设置项 `name` 的值

```
app.get('title');
// => undefined

app.set('title', 'My Site');
app.get('title');
// => "My Site"
```

app.enable(name)

将设置项 `name` 的值设为 `true` .

```
app.enable('trust proxy');
app.get('trust proxy');
// => true
```

app.disable(name)

将设置项 `name` 的值设为 `false`.

```
app.disable('trust proxy');
app.get('trust proxy');
// => false
```

app.enabled(name)

检查设置项 `name` 是否已启用

```
app.enabled('trust proxy');
// => false

app.enable('trust proxy');
app.enabled('trust proxy');
// => true
```

app.disabled(name)

检查设置项 `name` 是否已禁用

```
app.disabled('trust proxy');
// => true

app.enable('trust proxy');
app.disabled('trust proxy');
// => false
```

app.configure([env], callback)

当 `env` 和 `app.get('env')` (也就是 `process.env.NODE_ENV`) 匹配时, 调用 `callback` 。保留这个方法是出于历史原因，后面列出的 `if` 语句的代码其实更加高效、直接。使用 `app.set()` 配合其它一些配置方法后,没有必要再使用这个方法。

```
// 所有环境
app.configure(function(){
  app.set('title', 'My Application');
})

// 开发环境
app.configure('development', function(){
  app.set('db uri', 'localhost/dev');
})

// 只用于生产环境
app.configure('production', function(){
  app.set('db uri', 'n.n.n.n/prod');
})
```

更高效且直接的代码如下：

```
// 所有环境
app.set('title', 'My Application');

// 只用于开发环境
if ('development' == app.get('env')) {
  app.set('db uri', 'localhost/dev');
}

// 只用于生产环境
if ('production' == app.get('env')) {
  app.set('db uri', 'n.n.n.n/prod');
}
```

app.use([path], function)

使用中间件 `function`,可选参数 `path` 默认为"/"。

```
var express = require('express');
var app = express();

// 一个简单的 logger
app.use(function(req, res, next){
  console.log('%s %s', req.method, req.url);
  next();
});

// 响应
app.use(function(req, res, next){
  res.send('Hello World');
});

app.listen(3000);
```

挂载的路径不会在req里出现，对中间件 `function` 不可见，这意味着你在 `function` 的回调参数req里找不到path。这么设计的为了让间件可以在不需要更改代码就在任意"前缀"路径下执行

这里有一个实际应用场景，常见的一个应用是使用./public提供静态文件服务，用 `express.static()` 中间件:

```
// GET /javascripts/jquery.js
// GET /style.css
// GET /favicon.ico
app.use(express.static(__dirname + '/public'));
```

如果你想把所有的静态文件路径都前缀"/static", 你可以使用“挂载”功能。如果 `req.url` 不包含这个前缀, 挂载过的中间件不会执行。当 `function` 被执行的时候,这个参数不会被传递。这个只会影响这个函数，后面的中间件里得到的 `req.url` 里将会包含"/static"

```
// GET /static/javascripts/jquery.js
// GET /static/style.css
// GET /static/favicon.ico
app.use('/static', express.static(__dirname + '/public'));
```

使用 `app.use()` “定义的”中间件的顺序非常重要，它们将会顺序执行，use的先后顺序决定了中间件的优先级。比如说通常 `express.logger()` 是最先使用的一个组件，纪录每一个请求

```
app.use(express.logger());
app.use(express.static(__dirname + '/public'));
app.use(function(req, res){
  res.send('Hello');
});
```

如果你想忽略请求静态文件的纪录，但是对于在 `logger()` 之后定义的路由和中间件想继续纪录，只需要简单的把 `static()` 移到前面就行了:

```
app.use(express.static(__dirname + '/public'));
app.use(express.logger());
app.use(function(req, res){
  res.send('Hello');
});
```

另一个现实的例子，有可能从多个目录提供静态文件服务，下面的例子中会优先从"./public"目录取文件

```
app.use(express.static(__dirname + '/public'));
app.use(express.static(__dirname + '/files'));
app.use(express.static(__dirname + '/uploads'));
```

settings

下面的内建的可以改变Express行为的设置

- `env` 运行时环境，默认为 `process.env.NODE_ENV` 或者 "development"
- `trust proxy` 激活反向代理，默认未激活状态
- `jsonp callback name` 修改默认 `?callback=` 的jsonp回调的名字
- `json replacer` JSON replacer 替换时的回调, 默认为null
- `json spaces` JSON 响应的空格数量，开发环境下是 `2` , 生产环境是 `0`
- `case sensitive routing` 路由的大小写敏感, 默认是关闭状态， "/Foo" 和"/foo" 是一样的
- `strict routing` 路由的严格格式, 默认情况下 "/foo" 和 "/foo/" 是被同样对待的
- `view cache` 模板缓存，在生产环境中是默认开启的
- `view engine` 模板引擎
- `views` 模板的目录, 默认是"process.cwd() + ./views"

app.engine(ext, callback)

注册模板引擎的 `callback` 用来处理 `ext` 扩展名的文件 默认情况下, 根据文件扩展名 `require()` 对应的模板引擎。 比如你想渲染一个 "foo.jade" 文件，Express 会在内部执行下面的代码，然后会缓存 `require()` ，这样就可以提高后面操作的

性能

```
app.engine('jade', require('jade').__express);
```

那些没有提供 `.__express` 的或者你想渲染一个文件的扩展名与模板引擎默认的不一致的时候，也可以用这个方法。比如你想用EJS模板引擎来处理 ".html" 后缀的文件:

```
app.engine('html', require('ejs').renderFile);
```

这个例子中EJS提供了一个 `.renderFile()` 方法和Express预期的格式: `(path, options, callback)` 一致, 可以在内部给这个方法取一个别名 `ejs.__express` ，这样你就可以使用".ejs" 扩展而不需要做任何改动

有些模板引擎没有遵循这种转换，这里有一个小项目consolidate.js 专门把所有的node流行的模板引擎进行了包装，这样它们在Express内部看起来就一样了。

```
var engines = require('consolidate');
app.engine('haml', engines.haml);
app.engine('html', engines.hogan);
```

app.param([name], callback)

路由参数的处理逻辑。比如当 `:user` 出现在一个路由路径中，你也许会自动载入加载用户的逻辑，并把它放置到 `req.user` , 或者校验一下输入的参数是否正确。

下面的代码片段展示了`callback`很像中间件，但是在参数里多加了一个值，这里名为 `id` . 它会尝试加载用户信息，然后赋值给 `req.user` , 否则就传递错误 `next(err)` .

```
app.param('user', function(req, res, next, id){
  User.find(id, function(err, user){
    if (err) {
      next(err);
    } else if (user) {
      req.user = user;
      next();
    } else {
      next(new Error('failed to load user'));
    }
  });
});
```

另外你也可以只传一个 `callback`，这样你就有机会改变 `app.param()` API. 比如express-params定义了下面的回调，这个允许你使用一个给定的正则去限制参数。

下面的这个例子有一点点高级，检查如果第二个参数是一个正则，返回一个很像上面的"user"参数例子行为的回调函数。

```
app.param(function(name, fn){
  if (fn instanceof RegExp) {
    return function(req, res, next, val){
      var captures;
      if (captures = fn.exec(String(val))) {
        req.params[name] = captures;
        next();
      } else {
        next('route');
      }
    }
  }
});
```

这个函数现在可以非常有效的用来校验参数，或者提供正则捕获后的分组。

```
app.param('id', /^\d+$/);

app.get('/user/:id', function(req, res){
  res.send('user ' + req.params.id);
});

app.param('range', /^(\w+)\.\.(\w+)?$/);

app.get('/range/:range', function(req, res){
  var range = req.params.range;
  res.send('from ' + range[1] + ' to ' + range[2]);
});
```

app.VERB(path, [callback...], callback)

`app.VERB()` 方法为Express提供路由方法, **VERB** 是指某一个HTTP 动作, 比如 `app.post()`。 可以提供多个callbacks,这多个callbacks都将会被平等对待 ， 它们的行为跟中间件一样，也有一个例外的情况，如果某一个callback执行了 `next('route')`，它后面的callback就被忽略。这种情形会应用在当满足一个路由前缀，但是不需要处理这个路由，于是把它向后传递。

下面的代码片段展示最简单的路由定义。Express 会把路径字符串转为正则表达式，然后在符合规则的请求到达时立即使用。 请求参数不会 被考虑进来，比如 "GET /" 会匹配下面的这个路由, 而"GET /?name=tobi"同样也会匹配。

```
app.get('/', function(req, res){
  res.send('hello world');
});
```

同样也可以使用正则表达式，并且它能够在你指定特定路径的时候发挥大作用。 比如下面的例子可以匹配"GET /commits/71dbb9c"， 同时也能匹配 "GET /commits/71dbb9c..4c084f9".

```
app.get(/^\/commits\/(\w+)(?:\.\.(\w+))?$/, function(req, res){
  var from = req.params[0];
  var to = req.params[1] || 'HEAD';
  res.send('commit range ' + from + '..' + to);
});
```

可以传递一些回调，这对复用一些加载资源、校验的中间件很有用。

```
app.get('/user/:id', user.load, function(){
  // ...
})
```

这些回调同样可以通过数组传递，简单的放置在数组中即可。

```
var middleware = [loadForum, loadThread];

app.get('/forum/:fid/thread/:tid', middleware, function(){
  // ...
})

app.post('/forum/:fid/thread/:tid', middleware, function(){
  // ...
})
```

app.all(path, [callback...], callback)

这个方法很像 `app.VERB()` ，但是它匹配所有的HTTP动作

这个方法在给特定前缀路径或者任意路径上处理时会特别有用。 比如你想把下面的路由放在所有其它路由之前，它需要所有从这个路由开始的加载验证，并且自动加载一个用户 记住所有的回调都不应该被当作终点， `loadUser` 能够被当作一个任务，然后 `next()` 去匹配接下来的路由。

```
app.all('*', requireAuthentication, loadUser);
```

Or the equivalent:

```
app.all('*', requireAuthentication)
app.all('*', loadUser);
```

另一个非常赞的例子是全局白名单函数。这里有一个例子跟前一个很像，但是它限制前缀为"/api":

```
app.all('/api/*', requireAuthentication);
```

app.locals

应用程序本地变量会附加给所有的在这个应用程序内渲染的模板。 这是一个非常有用的模板函数，就像应用程序级数据一样。

```
app.locals.title = 'My App';
app.locals.strftime = require('strftime');
```

`app.locals` 对象是一个JavaScript `Function` , 执行的时候它会把属性合并到它自身，提供了一种简单展示已有对象作为本地变量的方法

```
app.locals({
  title: 'My App',
  phone: '1-250-858-9990',
  email: 'me@myapp.com'
});

app.locals.title
// => 'My App'

app.locals.email
// => 'me@myapp.com'
```

`app.locals` 对象最终会是一个JavaScript函数对象，你不可以使用Functions和Objects内置的属性，比如 `name, apply, bind, call, arguments, length, constructor`

```
app.locals({name: 'My App'});

app.locals.name
// => 返回 'app.locals' 而不是 'My App' (app.locals 是一个函数 !)
// => 如果name变量用在一个模板里, 发返回一个 ReferenceError
```

全部的保留字列表可以在很多规范里找到。 JavaScript 规范 介绍了原来的属性，有一些还会被现代的JS引擎识别，EcmaScript 规范 在它的基础上，统一了值，添加了一些，删除了一些废弃的。如果感兴趣，可以看看Functions和Objects的属性值。

默认情况下Express只有一个应用程序级本地变量，它是 `settings` .

```
app.set('title', 'My App');
// 在view里使用 settings.title
```

app.render(view, [options], callback)

渲染 `view` , `callback` 用来处理返回的渲染后的字符串。 这个是 `res.render()` 的应用程序级版本，它们的行为是一样的。

```
app.render('email', function(err, html){
  // ...
});

app.render('email', { name: 'Tobi' }, function(err, html){
  // ...
});
```

app.routes

`app.routes` 对象存储了所有的被HTTP verb定义路由。 这个对象可以用在一些内部功能上，比如Express不仅用它来做路由分发，同时在没有 `app.options()` 定义的情况下用它来处理默认的<string>OPTIONS</string>行为。 你的应用程序或者框架也可以很轻松的通过在这个对象里移除路由来达到删除路由的目的。

```
console.log(app.routes)

{ get:
   [ { path: '/',
       method: 'get',
       callbacks: [Object],
       keys: [],
       regexp: /^\/\/?$/i },
   { path: '/user/:id',
       method: 'get',
       callbacks: [Object],
       keys: [{ name: 'id', optional: false }],
       regexp: /^\/user\/(?:([^\/]+?))\/?$/i } ],
delete:
   [ { path: '/user/:id',
       method: 'delete',
       callbacks: [Object],
       keys: [Object],
       regexp: /^\/user\/(?:([^\/]+?))\/?$/i } ] }
```

app.listen()

在给定的主机和端口上监听请求，这个和node的文档http.Server#listen()是一致的

```
var express = require('express');
var app = express();
app.listen(3000);
```

`express()` 返回的 `app` 实际上是一个JavaScript `Function` ,它被设计为传给node的http servers作为处理请求的回调函数。因为 `app` 不是从HTTP或者HTTPS继承来的，它只是一个简单的回调函数，你可以以同一份代码同时处理HTTP and HTTPS 版本的服务。

```
var express = require('express');
var https = require('https');
var http = require('http');
var app = express();

http.createServer(app).listen(80);
https.createServer(options, app).listen(443);
```

`app.listen()` 方法只是一个快捷方法，如果你想使用HTTPS，或者同时提供HTTP和HTTPS，可以使用上面的代码

```
app.listen = function(){
  var server = http.createServer(this);
  return server.listen.apply(server, arguments);
};
```

Request

req.params

这是一个数组对象，命名过的参数会以键值对的形式存放。 比如你有一个路由 `/user/:name` ,"name"属性会存放在 `req.params.name` . 这个对象默认为 `{}` .

```
// GET /user/tj
req.params.name
// => "tj"
```

当使用正则表达式定义路由的时候， `req.params[N]` 会是这个应用这个正则后的捕获分组, `N` 是代表的是第N个捕获分组。这个规则同样适用于全匹配的路由，如 `/file/*` :

```
// GET /file/javascripts/jquery.js
req.params[0]
// => "javascripts/jquery.js"
```

req.query

这是一个解析过的请求参数对象，默认为 `{}` .

```
// GET /search?q=tobi+ferret
req.query.q
// => "tobi ferret"

// GET /shoes?order=desc&shoe[color]=blue&shoe[type]=converse
req.query.order
// => "desc"

req.query.shoe.color
// => "blue"

req.query.shoe.type
// => "converse"
```

req.body

这个对应的是解析过的请求体。这个特性是 `bodyParser()` 中间件提供,其它的请求体解析中间件可以放在这个中间件之后。当 `bodyParser()` 中间件使用后，这个对象默认为 `{}` 。

```
// POST user[name]=tobi&user[email]=tobi@learnboost.com
req.body.user.name
// => "tobi"

req.body.user.email
// => "tobi@learnboost.com"

// POST { "name": "tobi" }
req.body.name
// => "tobi"
```

req.files

这是上传的文件的对象。这个特性是 `bodyParser()` 中间件提供,其它的请求体解析中间件可以放在这个中间件之后。当 `bodyParser()` 中间件使用后，这个对象默认为 `{}` 。

例如 **file** 字段被命名为"image", 当一个文件上传完成后， `req.files.image` 将会包含下面的 `File` 对象:

```
{ size: 74643,
  path: '/tmp/8ef9c52abe857867fd0a4e9a819d1876',
  name: 'edge.png',
  type: 'image/png',
  hash: false,
  lastModifiedDate: Thu Aug 09 2012 20:07:51 GMT-0700 (PDT),
  _writeStream:
   { path: '/tmp/8ef9c52abe857867fd0a4e9a819d1876',
     fd: 13,
     writable: false,
     flags: 'w',
     encoding: 'binary',
     mode: 438,
     bytesWritten: 74643,
     busy: false,
     _queue: [],
     _open: [Function],
     drainable: true },
  length: [Getter],
  filename: [Getter],
  mime: [Getter] }
```

`bodyParser()` 中间件是在内部使用node-formidable来处理文件请求，所以接收的参数是一致的。 举个例子，使用formidable的选项 `keepExtensions` , 它默认为 **false** , 在上面的例子可以看到给出的文件名"/tmp/8ef9c52abe857867fd0a4e9a819d1876" 不包含".png" 扩展名. 为了让它可以保留扩展名，你可以把参数传给 `bodyParser()` :

```
app.use(express.bodyParser({ keepExtensions: true, uploadDir: '/my/
```

req.param(name)

返回 `name` 参数的值。

```
// ?name=tobi
req.param('name')
// => "tobi"

// POST name=tobi
req.param('name')
// => "tobi"

// /user/tobi for /user/:name
req.param('name')
// => "tobi"
```

查找的优先级如下:

- `req.params`
- `req.body`
- `req.query`

直接访问 `req.body` , `req.params` , 和 `req.query` 应该更合适，除非你真的需要从这几个对象里同时接受输入。

req.route

这个对象里是当前匹配的 `Route` 里包含的属性，比如原始路径字符串，产生的正则，等等

```
app.get('/user/:id?', function(req, res){
  console.log(req.route);
});
```

上面代码的一个输出:

```
{ path: '/user/:id?',
  method: 'get',
  callbacks: [ [Function] ],
  keys: [ { name: 'id', optional: true } ],
  regexp: /^\/user(?:\/([^\/]+?))?\/?$/i,
  params: [ id: '12' ] }
```

req.cookies

当使用 `cookieParser()` 中间件之后，这个对象默认为 `{}`，它也包含了用户代理传过来的cookies。

```
// Cookie: name=tj
req.cookies.name
// => "tj"
```

req.signedCookies

当使用了 `cookieParser(secret)` 中间件后，这个对象默认为 `{}`，否则包含了用户代理传回来的签名后的cookie，并等待使用。签名后的cookies被放在一个单独的对象里，恶意攻击者可以很简单的替换掉 `req.cookie` 的值。需要注意的是签名的cookie不代表它是隐藏的或者加密的，这个只是简单的阻止篡改cookie。

```
// Cookie: user=tobi.CP7AWaXDfAKIRfH49dQzKJx7sKzzSoPq7/AcBBRVwlI3
req.signedCookies.user
// => "tobi"
```

req.get(field)

获取请求头里的 `field` 的值，大小写不敏感. *Referrer* 和 *Referer* 字段是可以互换的。

```
req.get('Content-Type');
// => "text/plain"

req.get('content-type');
// => "text/plain"

req.get('Something');
// => undefined
```

别名为 `req.header(field)`.

req.accepts(types)

. 检查给定的 `types` 是不是可以接受类型，当可以接受时返回最匹配的，否则返回 `undefined` - 这个时候你应该响应一个406 "Not Acceptable".

`type` 的值可能是单一的一个mime类型字符串,比如 "application/json", 扩展名为"json", 也可以为逗号分隔的列表或者数组。当给定的是数组或者列表，返回最佳匹配的。

```
// Accept: text/html
req.accepts('html');
// => "html"

// Accept: text/*, application/json
req.accepts('html');
// => "html"
req.accepts('text/html');
// => "text/html"
req.accepts('json, text');
// => "json"
req.accepts('application/json');
// => "application/json"

// Accept: text/*, application/json
req.accepts('image/png');
req.accepts('png');
// => undefined

// Accept: text/*;q=.5, application/json
req.accepts(['html', 'json']);
req.accepts('html, json');
// => "json"
```

req.accepted

返回一个从高质量到低质量排序的接受媒体类型数组

```
[ { value: 'application/json',
    quality: 1,
    type: 'application',
    subtype: 'json' },
{ value: 'text/html',
     quality: 0.5,
     type: 'text',
     subtype: 'html' } ]
```

req.is(type)

检查请求的文件头是不是包含"Content-Type" 字段, 它匹配给定的 `type` .

```
// With Content-Type: text/html; charset=utf-8
req.is('html');
req.is('text/html');
req.is('text/*');
// => true

// When Content-Type is application/json
req.is('json');
req.is('application/json');
req.is('application/*');
// => true

req.is('html');
// => false
```

req.ip

返回远程地址，或者当“信任代理”使用时，返回上一级的地址

```
req.ip
// => "127.0.0.1"
```

req.ips

当设置"trust proxy" 为 `true` 时, 解析"X-Forwarded-For" 里的ip地址列表，并返回一个数组 否则返回一个空数组 举个例子，如果"X-Forwarded-For" 的值为"client, proxy1, proxy2" 你将会得到数组 `["client", "proxy1", "proxy2"]` 这里可以看到 "proxy2" 是最近一个使用的代理

req.path

返回请求的URL的路径名

```
// example.com/users?sort=desc
req.path
// => "/users"
```

req.host

返回从"Host"请求头里取的主机名,不包含端口号。

```
// Host: "example.com:3000"
req.host
// => "example.com"
```

req.fresh

判断请求是不是新的-通过对Last-Modified 或者 ETag 进行匹配, 来标明这个资源是不是"新的".

```
req.fresh
// => true
```

req.stale

判断请求是不是旧的-如果Last-Modified 或者 ETag 不匹配, 标明这个资源是"旧的". Check if the request is stale - aka Last-Modified and/or the ETag do not match, indicating that the resource is "stale".

```
req.stale
// => true
```

req.xhr

判断请求头里是否有"X-Requested-With"这样的字段并且值为"XMLHttpRequest", jQuery等库发请求时会设置这个头

```
req.xhr
// => true
```

req.protocol

返回标识请求协议的字符串，一般是"http"，当用TLS请求的时候是"https"。当"trust proxy" 设置被激活， "X-Forwarded-Proto" 头部字段会被信任。 如果你使用了一个支持https的反向代理，那这个可能是激活的。

```
req.protocol
// => "http"
```

req.secure

检查TLS 连接是否已经建立。 这是下面的缩写:

```
'https' == req.protocol;
```

req.subdomains

把子域当作一个数组返回

```
// Host: "tobi.ferrets.example.com"
req.subdomains
// => ["ferrets", "tobi"]
```

req.originalUrl

这个属性很像 `req.url` ,但是它保留了原始的url。 这样你在做内部路由的时候可以重写 `req.url` 。 比如app.use()的挂载功能会重写 `req.url` ， 把从它挂载的点开始

```
// GET /search?q=something
req.originalUrl
// => "/search?q=something"
```

req.acceptedLanguages

返回一个从高质量到低质量排序的接受语言数组

```
Accept-Language: en;q=.5, en-us
// => ['en-us', 'en']
```

req.acceptedCharsets

返回一个从高质量到低质量排序的可接受的字符集数组

```
Accept-Charset: iso-8859-5;q=.2, unicode-1-1;q=0.8
// => ['unicode-1-1', 'iso-8859-5']
```

req.acceptsCharset(charset)

检查给定的 `charset` 是不是可以接受的

req.acceptsLanguage(lang)

检查给定的 `lang` 是不是可以接受的

Response

res.status(code)

支持链式调用的 node's `res.statusCode=` .

```
res.status(404).sendfile('path/to/404.png');
```

res.set(field, [value])

设置响应头字段`field` 值为 `value`, 也可以一次传入一个对象设置多个值。

```
res.set('Content-Type', 'text/plain');

res.set({
  'Content-Type': 'text/plain',
  'Content-Length': '123',
  'ETag': '12345'
})
```

`res.header(field, [value])`的别名。

res.get(field)

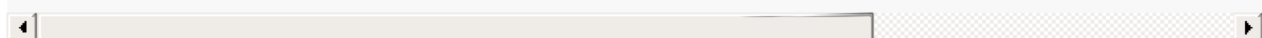
返回一个大小写不敏感的响应头里的 `field` 的值

```
res.get('Content-Type');
// => "text/plain"
```

res.cookie(name, value, [options])

设置cookie `name` 值为 `value`, 接受字符串参数或者JSON对象。 `path` 属性默认为 "/".

```
res.cookie('name', 'tobi', { domain: '.example.com', path: '/admin
res.cookie('rememberme', '1', { expires: new Date(Date.now() + 900(
```



`maxAge` 属性是一个便利的设置"expires",它是一个从当前时间算起的毫秒。 下面的代码和上一个例子中的第二行是同样的作用。

```
res.cookie('rememberme', '1', { maxAge: 900000, httpOnly: true })
```

可以传一个序列化的JSON对象作为参数, 它会自动被 `bodyParser()` 中间件解析。

```
res.cookie('cart', { items: [1,2,3] });
res.cookie('cart', { items: [1,2,3] }, { maxAge: 900000 });
```

这个方法也支持签名的cookies。 只需要简单的传递 `signed` 参数。 `res.cookie()` 会使用通过 `express.cookieParser(secret)` 传 入的secret来签名这个值

```
res.cookie('name', 'tobi', { signed: true });
```

稍后你就可以通过req.signedCookie 对象访问到这个值。

res.clearCookie(name, [options])

把 `name` 的cookie清除. `path` 参数默认为 "/".

```
res.cookie('name', 'tobi', { path: '/admin' });
res.clearCookie('name', { path: '/admin' });
```

res.redirect([status], url)

使用可选的状态码跳转到 `url` 状态码 `status` 默认为302 "Found".

```
res.redirect('/foo/bar');
res.redirect('http://example.com');
res.redirect(301, 'http://example.com');
res.redirect('../login');
```

Express支持几种跳转，第一种便是使用一个完整的URI跳转到一个完全不同的网站。

```
res.redirect('http://google.com');
```

第二种是相对根域路径跳转，比如你现在在 `http://example.com/admin/post/new`，下面的的代码跳转到 `/admin` 将会把你带到 `http://example.com/admin`：

```
res.redirect('/admin');
```

这是一种相对于应用程序挂载点的跳转。 比如把一个blog程序挂在 `/blog`，事实上它无法知道它被挂载，所以当你使用跳转 `/admin/post/new` 时，将到跳到 `http://example.com/admin/post/new`，下面的相对于挂载点的跳转会把你带到 `http://example.com/blog/admin/post/new`：

```
res.redirect('admin/post/new');
```

路径名.跳转同样也是支持的。 比如你在 `http://example.com/admin/post/new`，下面的跳转会把你带到 `http//example.com/admin/post`：

```
res.redirect('..');
```

最后也是最特别的跳转是 `back` 跳转，它会把你带回Referer（也有可能是Referrer）的地址 当Referer丢失的时候默认为 `/`

```
res.redirect('back');
```

res.location

设置location 请求头.

```
res.location('/foo/bar');
res.location('foo/bar');
res.location('http://example.com');
res.location('../login');
res.location('back');
```

可以使用与 `res.redirect()` 里相同的 `urls` 。

举个例子，如果你的程序根地址是 `/blog`，下面的代码会把 `location` 请求头设置为 `/blog/admin`：

```
res.location('admin')
```

res.charset

设置字符集。默认为"utf-8"。

```
res.charset = 'value';
res.send('some html');
// => Content-Type: text/html; charset=value
```

res.send([body|status], [body])

发送一个响应。

```
res.send(new Buffer('whoop'));
res.send({ some: 'json' });
res.send('some html');
res.send(404, 'Sorry, we cannot find that!');
res.send(500, { error: 'something blew up' });
res.send(200);
```

这个方法在输出non-streaming响应的时候自动完成了大量有用的任务 比如如果在它前面没有定义Content-Length, 它会自动设置; 比如加一些自动的 *HEAD*; 比如对HTTP缓存的支持 .

当参数为一个 `Buffer` 时 Content-Type 会被设置为 "application/octet-stream" 除非它之前有像下面的代码：

```
res.set('Content-Type', 'text/html');
res.send(new Buffer('some html'));
```

当参数为一个 `String` 时 Content-Type 默认设置为"text/html":

```
res.send('some html');
```

当参数为 `Array` 或者 `Object` 时 Express 会返回一个 JSON :

```
res.send({ user: 'tobi' })
res.send([1,2,3])
```

最后一条当一个 `Number` 作为参数， 并且没有上面提到的任何一条在响应体里，Express会帮你设置一个响应体 比如200 会返回字符"OK", 404会返回"Not Found"等等.

```
res.send(200)
res.send(204)
res.send(500)
```

res.json([status|body], [body])

返回一个 JSON 响应。 当 `res.send()` 的参数是一个对象或者数组的时候， 会调用这个方法。 当然它也在复杂的空值(null, undefined, etc)JSON转换的时候很有用， 因为规范上这些对象不是合法的JSON。

```
res.json(null)
res.json({ user: 'tobi' })
res.json(500, { error: 'message' })
```

res.jsonp([status|body], [body])

返回一个支持JSONP的JSON响应。 Send a JSON response with JSONP support. 这个方法同样使用了 `res.json()` , 只是加了一个可以自定义的 JSONP 回调支持。

```
res.jsonp(null)
// => null

res.jsonp({ user: 'tobi' })
// => { "user": "tobi" }

res.jsonp(500, { error: 'message' })
// => { "error": "message" }
```

默认情况下JSONP 回调的函数名就是 `callback` 。 你可以通过jsonp callback name来修改这个值。 下面是一些使用JSONP的例子。

```
// ?callback=foo
res.jsonp({ user: 'tobi' })
// => foo({ "user": "tobi" })

app.set('jsonp callback name', 'cb');

// ?cb=foo
res.jsonp(500, { error: 'message' })
// => foo({ "error": "message" })
```

res.type(type)

设置 Sets the Content-Type to the mime lookup of `type`, or when "/" is present the Content-Type is simply set to this literal value.

```
res.type('.html');
res.type('html');
res.type('json');
res.type('application/json');
res.type('png');
```

`res.contentType(type)` 方法的别名。

res.format(object)

设置特定请求头的响应。 这个方法使用 `req.accepted` ， 这是一个通过质量值作为优先级顺序的数组， 第一个回调会被执行。 当没有匹配时，服务器返回一个 406 "Not Acceptable", 或者执行 `default` 回调

Content-Type 在callback 被选中执行的时候会被设置好, 如果你想改变它，可以在callback内使用 `res.set()` 或者 `res.type()` 等

下面的例子展示了在请求头设置为"application/json" 或者 "*/json*"的时候 会返回 *`{ "message": "hey" }`* 如果设置的是"*/*" 那么所有的返回都将是"hey"

```
res.format({
  'text/plain': function(){
    res.send('hey');
  },

  'text/html': function(){
    res.send('hey');
  },

  'application/json': function(){
    res.send({ message: 'hey' });
  }
});
```

除了使用标准的MIME 类型，你也可以使用扩展名来映射这些类型 下面是一个不太完整的实现：

```
res.format({
  text: function(){
    res.send('hey');
  },

  html: function(){
    res.send('hey');
  },

  json: function(){
    res.send({ message: 'hey' });
  }
});
```

res.attachment([filename])

设置响应头的Content-Disposition 字段值为 "attachment". 如果有 `filename` 参数，Content-Type 将会依据文件扩展名通过 `res.type()` 自动设置, 并且Content-Disposition的"filename="参数将会被设置

```
res.attachment();
// Content-Disposition: attachment

res.attachment('path/to/logo.png');
// Content-Disposition: attachment; filename="logo.png"
// Content-Type: image/png
```

res.sendfile(path, [options], [fn]])

`path` 所传输附件的路径。

它会根据文件的扩展名自动设置响应头里的Content-Type字段。 回调函数 `fn(err)` 在传输完成或者发生错误时会被调用执行。

Options:

- `maxAge` 毫秒，默认为0
- `root` 文件相对的路径

这个方法可以非常良好的支持有缩略图的文件服务。

```
app.get('/user/:uid/photos/:file', function(req, res){
  var uid = req.params.uid
    , file = req.params.file;

  req.user.mayViewFilesFrom(uid, function(yes){
    if (yes) {
      res.sendfile('/uploads/' + uid + '/' + file);
    } else {
      res.send(403, 'Sorry! you cant see that.');
    }
  });
});
```

res.download(path, [filename], [fn])

`path` 所需传输附件的路径, 通常情况下浏览器会弹出一个下载文件的窗口。 浏览器弹出框里的文件名和响应头里的Disposition "filename=" 参数是一致的, 你也可以通过传入 `filename` 来自由设置。

当在传输的过程中发生一个错误时，可选的回调函数 `fn` 会被调用执行。 这个方法使用res.sendfile()传输文件。

```
res.download('/report-12345.pdf');

res.download('/report-12345.pdf', 'report.pdf');

res.download('/report-12345.pdf', 'report.pdf', function(err){
  if (err) {
    // 处理错误，请牢记可能只有部分内容被传输，所以
    // 检查一下res.headerSent
  } else {
    // 减少下载的积分值之类的
  }
});
```

res.links(links)

合并给定的 `links` , 并且设置给响应头里的"Link" 字段.

```
res.links({
  next: 'http://api.example.com/users?page=2',
  last: 'http://api.example.com/users?page=5'
});
```

转换后:

```
Link: <http://api.example.com/users?page=2>; rel="next",
      <http://api.example.com/users?page=5>; rel="last"
```

res.locals

在某一次请求范围下的响应体的本地变量，只对此次请求期间的views可见。 另外这个API其实和 app.locals是一样的.

这个对象在放置请求级信息时非常有用，比如放置请求的路径名，验证过的用户，用户设置等等

```
app.use(function(req, res, next){
  res.locals.user = req.user;
  res.locals.authenticated = ! req.user.anonymous;
  next();
});
```

res.render(view, [locals], callback)

渲染 `view` , 同时向callback 传入渲染后的字符串。 callback如果不传的话，直接会把渲染后的字符串输出至请求方， 一般如果不需要再对渲染后的模板作操作，就不需要传callback。 当有错误发生时 `next(err)` 会被执行. 如果提供了callback参数，可能发生的错误和渲染的字符串都会被当作参数传入, 并且没有默认响应。

```
res.render('index', function(err, html){
  // ...
});

res.render('user', { name: 'Tobi' }, function(err, html){
  // ...
});
```

Middleware

basicAuth()

基本的认证中间件，在 `req.user` 里添加用户名

用户名和密码的例子:

```
app.use(express.basicAuth('username', 'password'));
```

校验回调:

```
app.use(express.basicAuth(function(user, pass){
  return 'tj' == user && 'wahoo' == pass;
}));
```

异步校验接受参数 `fn(err, user)` , 下面的例子 `req.user` 将会作为user对象传递.

```
app.use(connect.basicAuth(function(user, pass, fn){
  User.authenticate({ user: user, pass: pass }, fn);
}))
```

bodyParser()

支持 JSON, urlencoded和multipart requests的请求体解析中间件。 这个中间件是 `json()` , `urlencoded()` ,和 `multipart()` 这几个中间件的简单封装

```
app.use(express.bodyParser());

// 等同于:
app.use(express.json());
app.use(express.urlencoded());
app.use(express.multipart());
```

从安全上考虑，如果你的应用程序不需要文件上传功能，最好关闭它。我们只使用我们需要的中间件。例如：我们不使用 `bodyParser` 、 `multipart()` 这两个中间件。

```
app.use(express.json());
app.use(express.urlencoded());
```

如果你的应用程序需要使用文件上传，设置一下就行。 一个简单的介绍如何使用.

compress()

通过gzip / deflate压缩响应数据. 这个中间件应该放置在所有的中间件最前面以保证所有的返回都是被压缩的

```
app.use(express.logger());
app.use(express.compress());
app.use(express.methodOverride());
app.use(express.bodyParser());
```

cookieParser()

解析请求头里的Cookie, 并用cookie名字的键值对形式放在 `req.cookies` 你也可以通过传递一个 `secret` 字符串激活签名了的cookie

```
app.use(express.cookieParser());
app.use(express.cookieParser('some secret'));
```

cookieSession()

提供一个以cookie为基础的sessions, 设置在 `req.session` 里。 这个中间件有以下几个选项:

- `key` cookie 的名字，默认是 `connect.sess`
- `secret` prevents cookie tampering
- `cookie` session cookie 设置, 默认是 `{ path: '/', httpOnly: true, maxAge: null }`
- `proxy` 当设置安全cookies时信任反向代理 (通过 "x-forwarded-proto")

```
app.use(express.cookieSession());
```

清掉一个cookie, 只需要在响应前把null赋值给session:

```
req.session = null
```

csrf()

CSRF 防护中间件

默认情况下这个中间件会产生一个名为"_csrf"的标志，这个标志应该添加到那些需要服务器更改的请求里，可以放在一个表单的隐藏域，请求参数等。这个标志可以通过 `req.csrfToken()` 方法进行校验。

`bodyParser()` 中间件产生的 `req.body` , `query()` 产生的 `req.query` ,请求头里的"X-CSRF-Token"是默认的 `value` 函数检查的项

这个中间件需要session支持，因此它的代码应该放在 `session()` 之后.

directory()

文件夹服务中间件，用 `path` 提供服务。

```
app.use(express.directory('public'))
app.use(express.static('public'))
```

这个中间件接收如下参数：

- `hidden` 显示隐藏文件，默认为false.
- `icons` 显示图标，默认为false.
- `filter` 在文件上应用这个过滤函数。默认为false.

Jade 文档

Jade 是一个高性能的模板引擎，它深受Haml影响，它是用javascript实现的,并且可以供node使用.

翻译:草依山 翻译反馈 Fork me

特性

- 客户端支持
- 代码高可读
- 灵活的缩进
- 块展开
- 混合
- 静态包含
- 属性改写
- 安全，默认代码是转义的
- 运行时和编译时上下文错误报告
- 命令行下编译jade模板
- html 5 模式 (使用 *!!! 5* 文档类型)
- 在内存中缓存(可选)
- 合并动态和静态标签类
- 可以通过 *filters* 修改树
- 模板继承
- 原生支持 Express JS
- 通过 `each` 枚举对象、数组甚至是不能枚举的对象
- 块注释
- 没有前缀的标签
- AST filters
- 过滤器
  - :sass 必须已经安装sass.js
  - :less 必须已经安装less.js
  - :markdown 必须已经安装markdown-js 或者node-discount
  - :cdata
  - :coffeescript 必须已经安装coffee-script
- Vim Syntax
- TextMate Bundle
- Screencasts
- html2jade 转换器

其它实现

- php
- scala

- ruby

安装

通过 npm:

```
npm install jade
```

浏览器支持

把jade编译为一个可供浏览器使用的单文件，只需要简单的执行:

```
$ make jade.js
```

如果你已经安装了uglifyjs (`npm install uglify-js`)，你可以执行下面的命令它会生成所有的文件。其实每一个正式版本里都帮你做了这事。

```
$ make jade.min.js
```

默认情况下，为了方便调试Jade会把模板组织成带有形如 `__.lineno = 3` 的行号的形式。 在浏览器里使用的时候，你可以通过传递一个选项 `{ compileDebug: false }` 来去掉这个。 下面的模板

```
p Hello #{name}
```

会被翻译成下面的函数：

```
function anonymous(locals, attrs, escape, rethrow) {
  var buf = [];
  with (locals || {}) {
    var interp;
    buf.push('\n<p>Hello ' + escape((interp = name) == null ? '' :
  }
  return buf.join("");
}
```

通过使用Jade的 `./runtime.js` 你可以在浏览器使用这些预编译的模板而不需要使用Jade, 你只需要使用runtime.js里的工具函数, 它们会放在 `jade.attrs` , `jade.escape` 这些里。 把选项 `{ client: true }` 传递给

`jade.compile()` , Jade 会把这些帮助函数的引用放在 `jade.attrs` , `jade.escape` .

```
function anonymous(locals, attrs, escape, rethrow) {
  var attrs = jade.attrs, escape = jade.escape, rethrow = jade.reth
  var buf = [];
  with (locals || {}) {
    var interp;
    buf.push('\n<p>Hello ' + escape((interp = name) == null ? '' :
  }
  return buf.join("");
}
```

公开API

```
var jade = require('jade');

   // Compile a function
   var fn = jade.compile('string of jade', options);
   fn(locals);
```

选项

- `self` 使用 `self` 命名空间来持有本地变量. 默认为*false*
- `locals` 本地变量对象
- `filename` 异常发生时使用，includes时必需
- `debug` 输出token和翻译后的函数体
- `compiler` 替换掉jade默认的编译器
- `compileDebug` `false` 的时候调试的结构不会被输出

语法

行结束标志

CRLF 和 **CR** 会在编译之前被转换为 **LF**

标签

标签就是一个简单的单词:

```
html
```

它会被转换为 `<html></html>`

标签也是可以有id的:

```
div#container
```

它会被转换为 `<div id="container"></div>`

怎么加类呢？

```
div.user-details
```

转换为 `<div class="user-details"></div>`

多个类? 和id? 也是可以搞定的:

```
div#foo.bar.baz
```

转换为 `<div id="foo" class="bar baz"></div>`

不停的div div div 很讨厌啊 , 可以这样:

```
#foo
.bar
```

这个算是我们的语法糖，它已经被很好的支持了，上面的会输出：

```
`<div id="foo"></div><div class="bar"></div>`
```

标签文本

只需要简单的把内容放在标签之后：

```
p wahoo!
```

它会被渲染为 `<p>wahoo!</p>` .

很帅吧，但是大段的文本怎么办呢：

```
p
  | foo bar baz
  | rawr rawr
  | super cool
  | go jade go
```

渲染为 `<p>foo bar baz rawr.....</p>`

怎么和数据结合起来？ 所有类型的文本展示都可以和数据结合起来，如果我们把 `{ name: 'tj', email: 'tj@vision-media.ca' }` 传给编译函数，下面是模板上的写法:

```
#user #{name} &lt;#{email}&gt;
```

它会被渲染为 `<div id="user">tj <tj@vision-media.ca></div>`

当就是要输出 `#{}` 的时候怎么办? 转义一下!

```
p \#{something}
```

它会输出 `<p>#{something}</p>`

同样可以使用非转义的变量 `!{html}` , 下面的模板将直接输出一个script标签

```
- var html = "<script></script>"
| !{html}
```

内联标签同样可以使用文本块来包含文本：

```
label
  | Username:
  input(name='user[name]')
```

或者直接使用标签文本:

```
label Username:
  input(name='user[name]')
```

只包含文本的标签，比如 `script` , `style` , 和 `textarea` 不需要前缀 `|` 字符, 比如:

```
html
   head
     title Example
     script
       if (foo) {
         bar();
       } else {
         baz();
       }
```

这里还有一种选择，可以使用'.' 来开始一段文本块，比如：

```
p.
   foo asdf
   asdf
    asdfasdfaf
    asdf
   asd.
```

会被渲染为:

```
<p>foo asdf
   asdf
     asdfasdfaf
     asdf
   asd
   .
   </p>
```

这和带一个空格的 '.' 是不一样的，带空格的会被Jade的解析器忽略，当作一个普通的文字:

```
p .
```

渲染为:

```
<p>.</p>
```

需要注意的是广西块需要两次转义。比如想要输出下面的文本：

```
</p>foo\bar</p>
```

使用:

```
p.
  foo\\bar
```

注释

单行注释和JavaScript里是一样的，通过"//"来开始，并且必须单独一行：

```
// just some paragraphs
p foo
p bar
```

渲染为：

```
<!-- just some paragraphs -->
<p>foo</p>
<p>bar</p>
```

Jade 同样支持不输出的注释，加一个短横线就行了：

```
//- will not output within markup
p foo
p bar
```

渲染为：

```
<p>foo</p>
<p>bar</p>
```

块注释

块注释也是支持的：

```
 body
    //
      #content
        h1 Example
```

渲染为：

```
<body>
  <!--
  <div id="content">
    <h1>Example</h1>
  </div>
  -->
</body>
```

Jade 同样很好的支持了条件注释：

```
body
  //if IE
    a(href='http://www.mozilla.com/en-US/firefox/') Get Firefox
```

渲染为：

内联

Jade 支持以自然的方式定义标签嵌套:

```
ul
  li.first
    a(href='#') foo
  li
    a(href='#') bar
  li.last
    a(href='#') baz
```

块展开

块展开可以帮助你在一行内创建嵌套的标签，下面的例子和上面的是一样的：

```
 ul
    li.first: a(href='#') foo
    li: a(href='#') bar
    li.last: a(href='#') baz
```

属性

Jade 现在支持使用'(' 和 ')' 作为属性分隔符

```
a(href='/login', title='View login page') Login
```

当一个值是 `undefined` 或者 `null` 属性不会被加上, 所以呢，它不会编译出 'something="null"'.

```
div(something=null)
```

Boolean 属性也是支持的:

```
input(type="checkbox", checked)
```

使用代码的Boolean 属性只有当属性为 `true` 时才会输出：

```
input(type="checkbox", checked=someValue)
```

多行同样也是可用的：

```
input(type='checkbox',
  name='agreement',
  checked)
```

多行的时候可以不加逗号：

```
input(type='checkbox'
  name='agreement'
  checked)
```

加点空格，格式好看一点？同样支持

```
input(
  type='checkbox'
  name='agreement'
  checked)
```

冒号也是支持的:

```
rss(xmlns:atom="atom")
```

假如我有一个 `user` 对象 `{ id: 12, name: 'tobi' }` 我们希望创建一个指向"/user/12"的链接 `href` , 我们可以使用普通的javascript字符串连接，如下:

```
a(href='/user/' + user.id)= user.name
```

或者我们使用jade的修改方式,这个我想很多使用Ruby或者 CoffeeScript的人会看起来像普通的js..:

a(href='/user/#{user.id}')= user.name

`class` 属性是一个特殊的属性，你可以直接传递一个数组，比如 `bodyClasses = ['user', 'authenticated']` ：

```
body(class=bodyClasses)
```

HTML

内联的html是可以的，我们可以使用管道定义一段文本：

```
html
  body
    | <h1>Title</h1>
    | <p>foo bar baz</p>
```

或者我们可以使用 `.` 来告诉Jade我们需要一段文本：

```
html
  body.
    <h1>Title</h1>
    <p>foo bar baz</p>
```

上面的两个例子都会渲染成相同的结果：

```
<html><body><h1>Title</h1>
<p>foo bar baz</p>
</body></html>
```

这条规则适应于在jade里的任何文本：

```
html
  body
    h1 User <em>#{name}</em>
```

Doctypes

添加文档类型只需要简单的使用 `!!!` , 或者 `doctype` 跟上下面的可选项:

```
!!!
```

会渲染出 *transitional* 文档类型, 或者:

```
!!! 5
```

or

```
!!! html
```

or

```
doctype html
```

doctypes 是大小写不敏感的, 所以下面两个是一样的:

```
doctype Basic
doctype basic
```

当然也是可以直接传递一段文档类型的文本：

```
doctype html PUBLIC "-//W3C//DTD XHTML Basic 1.1//EN
```

渲染后:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Basic 1.1//EN>
```

会输出 *html 5* 文档类型. 下面的默认的文档类型，可以很简单的扩展：

```
var doctypes = exports.doctypes = {
        '5': '<!DOCTYPE html>',
        'xml': '<?xml version="1.0" encoding="utf-8" ?>',
        'default': '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Tr
        'transitional': '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1
        'strict': '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Str
        'frameset': '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 F
        '1.1': '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" '
        'basic': '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Basic 1.
        'mobile': '<!DOCTYPE html PUBLIC "-//WAPFORUM//DTD XHTML Mo
    };
```

通过下面的代码可以很简单的改变默认的文档类型：

```
jade.doctypes.default = 'whatever you want';
```

过滤器

过滤器前缀 `:` , 比如 `:markdown` 会把下面块里的文本交给专门的函数进行处理。查看顶部 特性 里有哪些可用的过滤器。

```
body
  :markdown
    Woah! jade _and_ markdown, very **cool**
    we can even link to [stuff](http://google.com)
```

渲染为:

```
<body><p>Woah! jade <em>and</em> markdown, very <strong>cool</stro
```

代码

Jade目前支持三种类型的可执行代码。第一种是前缀 `-` ， 这是不会被输出的：

```
- var foo = 'bar';
```

这可以用在条件语句或者循环中：

```
- for (var key in obj)
  p= obj[key]
```

由于Jade的缓存技术，下面的代码也是可以的：

```
- if (foo)
  ul
    li yay
    li foo
    li worked
- else
  p oh no! didnt work
```

哈哈，甚至是很长的循环也是可以的：

```
- if (items.length)
  ul
    - items.forEach(function(item){
      li= item
    - })
```

所以你想要的！

下一步我们要转义输出的代码，比如我们返回一个值，只要前缀一个 `=` ：

```
- var foo = 'bar'
= foo
h1= foo
```

它会渲染为 `bar<h1>bar</h1>` . 为了安全起见，使用 `=` 输出的代码默认是转义的,如果想直接输出不转义的值可以使用 `!=` ：

```
p!= aVarContainingMoreHTML
```

Jade 同样是设计师友好的，它可以使javascript更直接更富表现力。比如下面的赋值语句是相等的，同时表达式还是通常的javascript：

```
 - var foo = 'foo ' + 'bar'
 foo = 'foo ' + 'bar'
```

Jade会把 `if` , `else if` , `else` , `until` , `while` , `unless` 同别的优先对待，但是你得记住它们还是普通的javascript：

```
if foo == 'bar'
  ul
    li yay
    li foo
    li worked
else
  p oh no! didnt work
```

循环

尽管已经支持JavaScript原生代码，Jade还是支持了一些特殊的标签，它们可以让模板更加易于理解，其中之一就是 `each` ，这种形式：

```
each VAL[, KEY] in OBJ
```

一个遍历数组的例子 ：

```
- var items = ["one", "two", "three"]
each item in items
  li= item
```

渲染为:

```
<li>one</li>
<li>two</li>
<li>three</li>
```

遍历一个数组同时带上索引：

```
items = ["one", "two", "three"]
each item, i in items
  li #{item}: #{i}
```

渲染为:

```
<li>one: 0</li>
<li>two: 1</li>
<li>three: 2</li>
```

遍历一个数组的键值 ：

```
obj = { foo: 'bar' }
each val, key in obj
  li #{key}: #{val}
```

将会渲染为： `<li>foo: bar</li>`

Jade在内部会把这些语句转换成原生的JavaScript语句，就像使用 `users.forEach(function(user){` , 词法作用域和嵌套会像在普通的JavaScript中一样：

```
each user in users
  each role in user.roles
    li= role
```

如果你喜欢，也可以使用 `for` ：

```
for user in users
  for role in user.roles
    li= role
```

条件语句

Jade 条件语句和使用了(`-`) 前缀的JavaScript语句是一致的,然后它允许你不使用圆括号，这样会看上去对设计师更友好一点， 同时要在心里记住这个表达式渲染出的是_常规_Javascript：

```
for user in users
  if user.role == 'admin'
    p #{user.name} is an admin
  else
    p= user.name
```

和下面的使用了常规JavaScript的代码是相等的：

```
 for user in users
   - if (user.role == 'admin')
     p #{user.name} is an admin
   - else
     p= user.name
```

Jade 同时支持 `unless` , 这和 `if (!(expr))` 是等价的：

```
for user in users
  unless user.isAnonymous
    p
      | Click to view
      a(href='/users/' + user.id)= user.name
```

模板继承

Jade 支持通过 `block` 和 `extends` 关键字来实现模板继承。 一个块就是一个Jade的"block" ， 它将在子模板中实现，同时是支持递归的。

Jade 块如果没有内容，Jade会添加默认内容，下面的代码默认会输出 `block scripts` , `block content` , 和 `block foot` .

```
html
  head
    h1 My Site - #{title}
    block scripts
      script(src='/jquery.js')
  body
    block content
    block foot
      #footer
        p some footer content
```

现在我们来继承这个布局，简单创建一个新文件，像下面那样直接使用 `extends` ，给定路径（可以选择带.jade扩展名或者不带）. 你可以定义一个或者更多的块来覆盖父级块内容, 注意到这里的 `foot` 块没有定义，所以它还会输出父级的"some footer content"。

```
extends extend-layout

block scripts
  script(src='/jquery.js')
  script(src='/pets.js')

block content
  h1= title
  each pet in pets
    include pet
```

同样可以在一个子块里添加块，就像下面实现的块 `content` 里又定义了两个可以被实现的块 `sidebar` 和 `primary` ，或者子模板直接实现 `content` 。

```
extends regular-layout

block content
  .sidebar
    block sidebar
      p nothing
  .primary
    block primary
      p nothing
```

包含

Includes 允许你静态包含一段Jade, 或者别的存放在单个文件中的东西比如css, html。 非常常见的例子是包含头部和页脚。 假设我们有一个下面目录结构的文件夹：

```
./layout.jade
./includes/
  ./head.jade
  ./tail.jade
```

下面是 *layout.jade* 的内容:

```
html
  include includes/head
  body
    h1 My Site
    p Welcome to my super amazing site.
    include includes/foot
```

这两个包含 *includes/head* 和 *includes/foot* 都会读取相对于给 *layout.jade* 参数 `filename` 的路径的文件, 这是一个绝对路径，不用担心Express帮你搞定这些了。Include 会解析这些文件，并且插入到已经生成的语法树中，然后渲染为你期待的内容：

```
<html>
  <head>
    <title>My Site</title>
    <script src="/javascripts/jquery.js">
    </script><script src="/javascripts/app.js"></script>
  </head>
  <body>
    <h1>My Site</h1>
    <p>Welcome to my super lame site.</p>
    <div id="footer">
      <p>Copyright>(c) foobar</p>
    </div>
  </body>
</html>
```

前面已经提到， `include` 可以包含比如html或者css这样的内容。给定一个扩展名后，Jade不会把这个文件当作一个Jade源代码，并且会把它当作一个普通文本包含进来：

```
html
  body
    include content.html
```

Include 也可以接受块内容，给定的块将会附加到包含文件 最后 的块里。 举个例子， `head.jade` 包含下面的内容：

```
head
  script(src='/jquery.js')
```

我们可以像下面给 `include head` 添加内容, 这里是添加两个脚本.

```
html
  include head
    script(src='/foo.js')
    script(src='/bar.js')
  body
    h1 test
```

Mixins

Mixins在编译的模板里会被Jade转换为普通的JavaScript函数。 Mixins 可以还参数，但不是必需的：

```
mixin list
   ul
     li foo
     li bar
     li baz
```

使用不带参数的mixin看上去非常简单，在一个块外：

```
h2 Groceries
 mixin list
```

Mixins 也可以带一个或者多个参数，参数就是普通的javascripts表达式，比如下面的例子：

```
mixin pets(pets)
   ul.pets
     - each pet in pets
       li= pet

 mixin profile(user)
   .user
     h2= user.name
     mixin pets(user.pets)
```

会输出像下面的html：

```
<div class="user">
  <h2>tj</h2>
  <ul class="pets">
    <li>tobi</li>
    <li>loki</li>
    <li>jane</li>
    <li>manny</li>
  </ul>
</div>
```

产生输出

假设我们有下面的Jade源码：

```
- var title = 'yay'
h1.title #{title}
p Just an example
```

当 `compileDebug` 选项不是 `false` , Jade 会编译时会把函数里加上 `__.lineno = n;` , 这个参数会在编译出错时传递给 `rethrow()` , 而这个函数会在Jade初始输出时给出一个有用的错误信息。

```
function anonymous(locals) {
  var __ = { lineno: 1, input: "- var title = 'yay'\nh1.title #{tit
  var rethrow = jade.rethrow;
  try {
    var attrs = jade.attrs, escape = jade.escape;
    var buf = [];
    with (locals || {}) {
      var interp;
      __.lineno = 1;
       var title = 'yay'
      __.lineno = 2;
      buf.push('<h1');
      buf.push(attrs({ "class": ('title') }));
      buf.push('>');
      buf.push('' + escape((interp = title) == null ? '' : interp)
      buf.push('</h1>');
      __.lineno = 3;
      buf.push('<p>');
      buf.push('Just an example');
      buf.push('</p>');
    }
    return buf.join("");
  } catch (err) {
    rethrow(err, __.input, __.filename, __.lineno);
  }
}
```

当 `compileDebug` 参数是 `false` , 这个参数会被去掉，这样对于轻量级的浏览器端模板是非常有用的。结合Jade的参数和当前源码库里的 `./runtime.js` 文件，你可以通过toString()来编译模板而不需要在浏览器端运行整个Jade库，这样可以提高性能，也可以减少载入的JavaScript数量。

```
function anonymous(locals) {
  var attrs = jade.attrs, escape = jade.escape;
  var buf = [];
  with (locals || {}) {
    var interp;
    var title = 'yay'
    buf.push('<h1');
    buf.push(attrs({ "class": ('title') }));
    buf.push('>');
    buf.push('' + escape((interp = title) == null ? '' : interp) +
    buf.push('</h1>');
    buf.push('<p>');
    buf.push('Just an example');
    buf.push('</p>');
  }
  return buf.join("");
}
```

Makefile的一个例子

通过执行 `make` ， 下面的Makefile例子可以把 *pages/*.jade* 编译为 *pages/*.html* 。

```
JADE = $(shell find pages/*.jade)
HTML = $(JADE:.jade=.html)

all: $(HTML)

%.html: %.jade
	jade < $< --path $< > $@

clean:
	rm -f $(HTML)

.PHONY: clean
```

这个可以和 `watch(1)` 命令起来产生像下面的行为：

```
 $ watch make
```

命令行的jade(1)

```
 使用: jade [options] [dir|file ...]

选项:

  -h, --help          输出帮助信息
  -v, --version       输出版本号
  -o, --obj <str>     javascript选项
  -O, --out <dir>     输出编译后的html到<dir>
  -p, --path <path>   在处理stdio时, 查找包含文件时的查找路径

Examples:

  # 编译整个目录
  $ jade templates

  # 生成 {foo,bar}.html
  $ jade {foo,bar}.jade

  # 在标准IO下使用jade
  $ jade < my.jade > my.html

  # 在标准IO下使用jade, 同时指定用于查找包含的文件
  $ jade < my.jade -p my.jade > my.html

  # 在标准IO下使用jade
  $ echo "h1 Jade!" | jade

  # foo, bar 目录渲染到 /tmp
  $ jade foo bar --out /tmp
```

License

Edit By MaHua